COMP 250

Lecture 3

binary numbers (continued),
Java primitive types,
ascii, Unicode
casting

Wed. Jan. 12, 2022

Addition in binary

The addition and multiplication algorithms from lecture 1 are based on operators $+, *, /, \%$. These algorithms work for any base. (See lecture notes.)

For example,

$$\begin{array}{r} 11010 \\ +\ \ 01111 \\ \hline ? \end{array} \qquad\qquad \begin{array}{r} 26 \\ +\ \ 15 \\ \hline 41 \end{array}$$

Addition in binary

The addition and multiplication algorithms from lecture 1 are based on operators $+$, $*$, $/$, $\%$. These algorithms work for any base. (See lecture notes.)

For example,

$$\begin{array}{lrcr} \text{carry} & 111100 & & \\ & 11010 & & 26 \\ + & 01111 & + & 15 \\ \hline & 101001 & & 41 \end{array}$$

Addition in binary

Fun example:

$$\begin{array}{r} 11111111111 \\ +\ \ \underline{00000000001} \\ 100000000000 \end{array}$$

[youtube video of odometer at 100,000]

Let’s use the example above to ask a fundamental question about binary number representations (next slide).

Q: How many bits *N* do we *need* to represent a positive integer m in binary ?

$$m = \sum_{i=0}^{N-1} b_i \, 2^i$$

What do we mean by “need”? We man using as few bits as possible, such that $b_{N-1} = 1$ and $b_i = 0$ for $i \geq N$.

For example, we consider representations like $(11010)_2$ but not $(000011010)_2$

Q: How many bits *N* do we *need* to represent a positive integer m in binary ?

Assuming we are using as few bits as possible, suppose:

$$m = (b_{N-1} \quad \dots \quad b_4 b_3 b_2 b_1 b_0)_2$$

The *smallest* that m can be is the N bit number:

$$(100000 \quad \dots \quad 000000)_2 = \ ?$$

The *largest* that m can be is the N bit number:

$$(111111 \quad \dots \quad 111111)_2 = \ ?$$

Q: How many bits *N* do we *need* to represent a positive integer m in binary ?

Assuming we are using as few bits as possible, suppose:

$$m = (b_{N-1} \quad \ldots \quad b_4 b_3 b_2 b_1 b_0)_2$$

The *smallest* that m can be is the N bit number:

$$(100000 \ \ldots \ 000000)_2 = 2^{N-1}$$

The *largest* that m can be is the N bit number:

$$(111111 \ \ldots \ 111111)_2 = 2^N - 1$$

From the previous slide: $2^{N-1} \leq m < 2^N$

Take the log (base 2) of each of the three terms :

$$N - 1 \leq log_2 m < N$$

The inequality is still correct since the log function is strictly increasing.

From here, we can show:

$N = floor(log_2 m) + 1$ where "floor" means "round down".

m (decimal)	m (binary)	$N = floor(\ log_2 m) + 1$
0	0	undefined
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4
10	1010	4
11	1011	4
:	:	:

Exact powers of 2 shown in red.

ASIDE: How are numbers represented in a computer?

How are integers represented (both positive and negative) ?

How are fractional numbers represented ?

Surprisingly, the answers do *not* depend on the computer or language.
Rather, there is a standard format that is used by all computers.
(For fractional numbers, the format is the IEEE 754 standard.)

The technical details are covered in detail in COMP 273 – see my lecture notes for that course if you are curious. I will say a bit about how integers are represented a few slides from now.

Java primitive types

`byte`	integer values
`short`	
`int`	
`long`	
`float`	fractional ("real") numbers
`double`	
`boolean`	`true` or `false`
`char`	One character

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

These are reserved keywords.

Java variables : type declaration

We can declare a primitive type variable as follows.

```
int       i;
double    x;
```

i ☐

x ☐

In order to use it, we need to assign it a value.

```
i = 3;
x = 4.75;
```

i 3

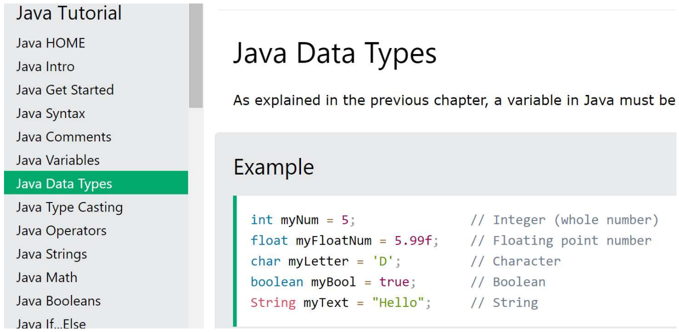
x 4.75

We can declare and assign a value in a single statement.

```
boolean  b = false;
```

b false

false is 00000000
true is 00000001



Java Data Types

As explained in the previous chapter, a variable in Java must be

Example

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99f;    // Floating point number
char myLetter = 'D';         // Character
boolean myBool = true;       // Boolean
String myText = "Hello";     // String
```

Java primitive types : what do they encode?

The number of bits used for each data type is fixed.

The bits can encode a particular set of values.

Keyword	Size	Values
`byte`	8-bits	
`short`	16-bits	
`int`	32-bits	
`long`	64-bits	
`float`	32-bits	
`double`	64-bits	
`boolean`	1-bit	
`char`	16-bits	

Java primitive types : what do they encode?

The number of bits used for each data type is fixed.

The bits can encode a particular set of values.

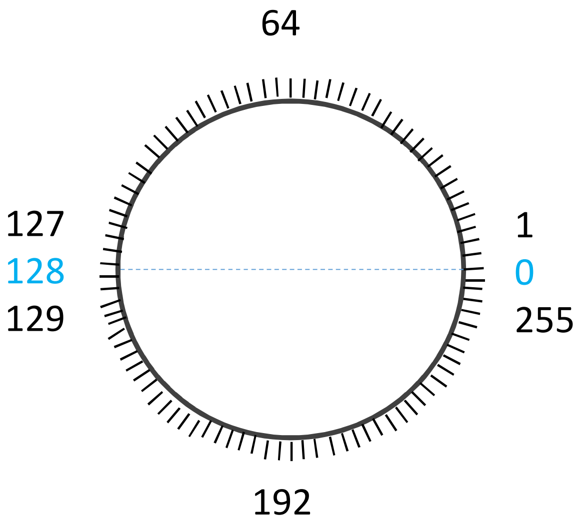
Keyword	Size	Values
`byte`	8-bits	$\{-2^7, \dots, 2^7 - 1\}$
`short`	16-bits	$\{-2^{15}, \dots, 2^{15} - 1\}$
`int`	32-bits	$\{-2^{31}, \dots, 2^{31} - 1\}$
`long`	64-bits	$\{-2^{63}, \dots, 2^{63} - 1\}$
`float`	32-bits	COMP 273/ECSE 222
`double`	64-bits	COMP 273/ECSE 222
`boolean`	1-bit	{true, false}
`char`	16-bits	later today

As I mentioned on slide 12, this uses 8 bits, but only the last bit matters

N bit integers

Recall the concept of modulus: like a circle, if you keep walking forward, you get back to where you started. With modulus operator, the circle is 0, 1, … , $2^{N} - 1$.

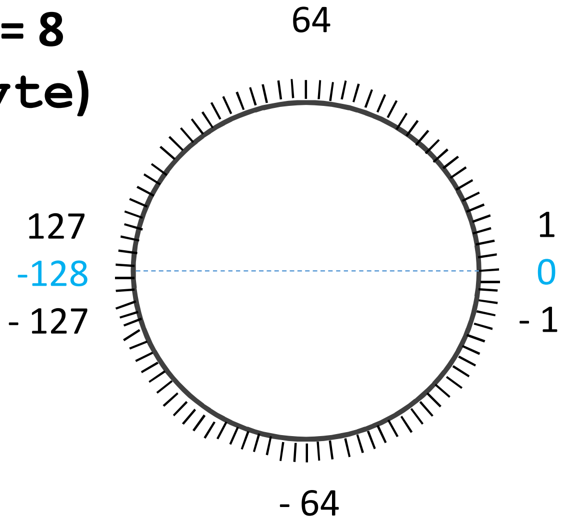
e.g. N = 8



Java represents “signed” integers. The values on the circle go from 0, 1, …. $2^{N-1} - 1$, and then jump back to -2^{N-1}, $-2^{N-1} + 1$, ….. -1.

e.g. N = 8

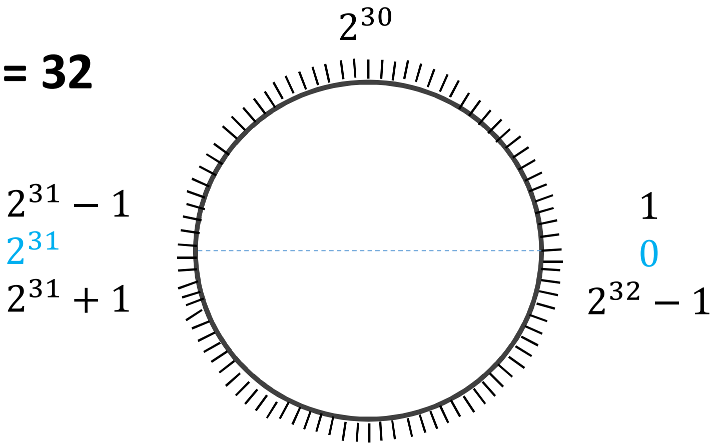
(`byte`)



N bit integers

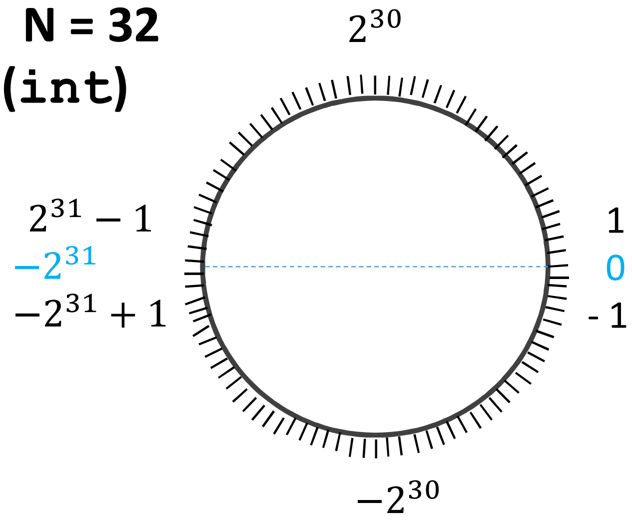
Recall the concept of modulus: like a circle, if you keep walking forward, you get back to where you started. With modulus operator, the circle is 0, 1, … , $2^{N} - 1$.

e.g. N = 32



Java represents “signed” integers. The values on the circle go from 0, 1, …. $2^{N-1} - 1$, and then jump back to -2^{N-1}, $-2^{N-1} + 1$, ….. -1.

e.g. N = 32
(`int`)

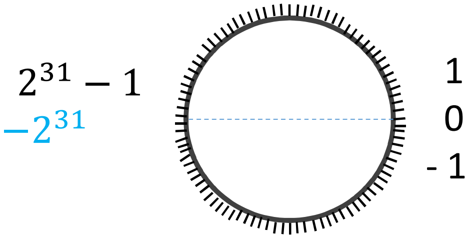


“Overflow” and “Underflow” e.g. `int`

Variables of type `int` store integer values from -2^{31} to $2^{31}-1$.

$-2^{31} = -2147483648$ ← min int value

$2^{31}-1 = 2147483647$ ← max int value



Example of overflow :

```
int x = 2147483647;          (max)
System.out.println(x+1);


→ prints   -2147483648        (min)
```

Example of underflow :

```
int y = -2147483648;         (min)
System.out.println(y-1);


→ prints   2147483647         (max)
```

Floating Point

- In Java, fractional numbers are represented using “floating point”, similar to scientific notation. e.g. 6.022149×10^{23}
- The type can be either `float` (32 bits) or `double` (64 bits).
- All standard arithmetic operations (+, -, *, /) can be done on floating point.
- Java distinguishes between `1` and `1.0`. If you write .0 after an integer, it will be represented as a `double`. If you want to represent a `float`, see below.

`int x = 3.0;` ✗

`int x = 3;` ✓

`double x = 3.0;` ✓

`float y = 3.0;` ✗

`float y = 3.0f;` ✓

Floating point approximation (round off)

The value of 1/3.0 is an approximation only.

More surprising perhaps, the value of 1/10.0 is also an approximation only.

The reason is that computers only represent sums of powers of 2*, including negative powers of 2* (1/2.0, 1/4.0, 1/8.0, etc).

Here is an interesting example:

```
System.out.println( 0.1 + 0.1 + 0.1 + 0.1 + 0.1 +
                    0.1 + 0.1 + 0.1 + 0.1 + 0.1  );
```

It prints out `0.9999999999999999` rather than `1.0`

`char` data type

We can declare and initialize a variable of type `char` as follows:

```
char letter = 'a';
```

- Character literals appears in single quotes.

  (A ‘literal’ is a particular character, string, or number.)

  Character literals can only contain a single character.   If you put two characters inside quotes, then it is not a character but rather it is a string.

Escape Sequences

An escape sequence is a sequence of characters that represents *a special character*. In Java, escape sequences are two characters and the first is a backslash.

Examples:

`\n` says to start a new line (e.g. when printing)
`\"` or `\'` represent quotation marks
`\t` represents a tab
`\\` represents a backslash

Escape sequences are legal characters. e.g. `char c = '\n';`

`char` data type

The `char` data type is two bytes (16 bits).

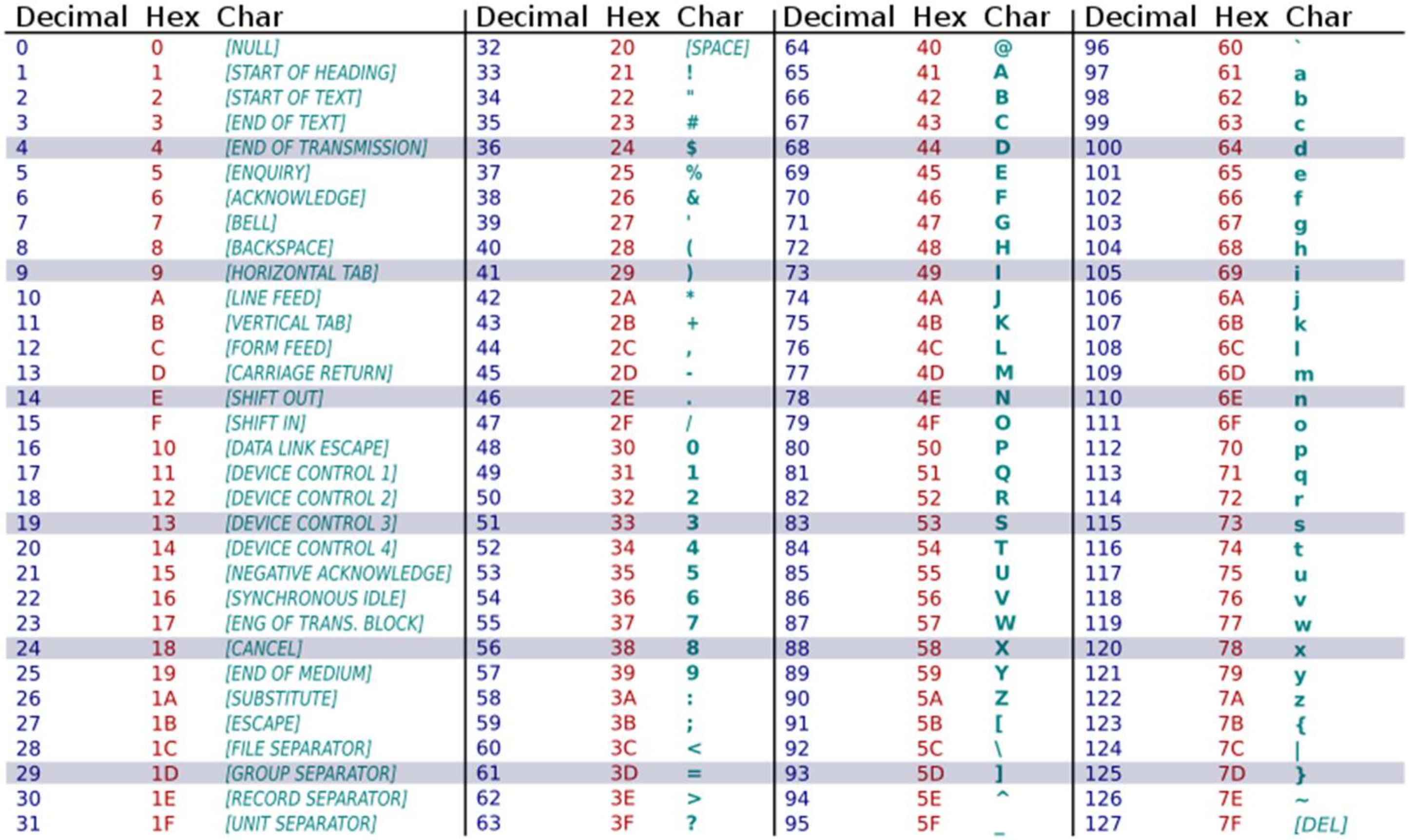
Think of them as numbered from 0 to $2^{16} - 1$ (65,535).

A common notation is `‘\u----’` where the `-` places are hexadecimal digits.

i.e. the values of a `char` range from `'\u0000'` to `'\uffff'`

The first $2^7 = 128$ of them correspond to the ASCII characters (next slide).

ASCII table



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Unicode

The `char` data uses Unicode which is an international standard.

Unicode is a superset of ASCII: the numbers 0-127 map to the same characters both in ASCII and Unicode.

Unicode provides the fonts for many languages. It also encodes emoji's.

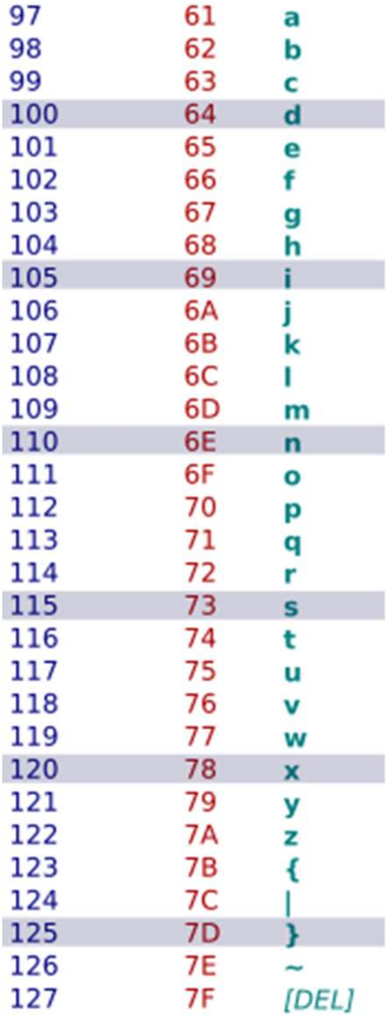
ASIDE: *there's more to Unicode than this:* you can expand beyond 2^{16} symbols by having pairs of `char` where the first one is essentially an escape character.

When we say that ASCII and Unicode are “codes”, we mean that each character or symbol is represented by a sequence of bits.

But we know sequences of bits also represent numbers!

In Java,   we can perform arithmetic with `char` values e.g.:

```
char c = 'a';      //  97
int  k = c + 1;    //  98
```



97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	[DEL]

Comparing Chars

We can also compare `char` values using operators $==$, $<$, $>$, $>=$, $<=$ which essentially compares their code values.

```
char letter0 = 'g';
char letter1 = 'k';
System.out.println( letter0 < letter1 );       //  prints true

System.out.println( 'g' < 'G' );               //  prints false

System.out.println( '%' >= '&' );              //  prints false
```

Type casting

Convert the values of variables from one type to another using **type casting**.

```
double y = 4.56;
int    n = (int) y;    //  value of n is ??????

int    x = 3;
double z = (double) x;    //  value of z is  ??????
```

Type casting

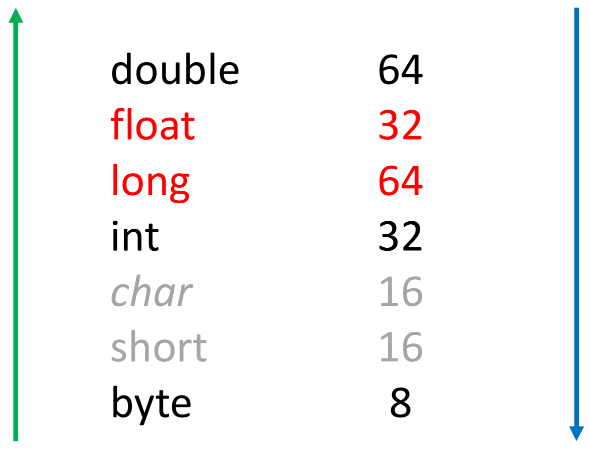
Convert the values of variables from one type to another using **type casting**.

```
double y = 4.56;
int    n = (int) y;   // value of n is 4 (rounds down)

int    x = 3;
double z = (double) x;          //  value of z is  3.0
```

As we will see next, an explicit cast is unnecessary here.

Primitive type conversion – wider vs. narrower



wider

implicit cast

"Wider" usually (but not always) means more bits.

type	*number of bits*
double	64
float	32
long	64
int	32
char	16
short	16
byte	8

narrower

explicit cast is needed

char and short are special ... see later.

Examples of widening & narrowing

```
int i = 3;
double d = 4.2;
```

`d = i;` widening ("implicit casting") → stores value 3.0

`i = d;` ✗ compile time error

`i = (int)  d;` narrowing ("explicit casting") → stores value 4

Examples of widening & narrowing

```
int i = 3;
double d = 4.2;

d = 5.3 * i;
```

widening by "promotion"

(the casting here happens when the * operation is performed)

```
byte k = 127;
System.out.println(k + 1);
```

widening by "promotion“ (to integer)

(the casting here happens when the + operation is performed)

Output: `128`

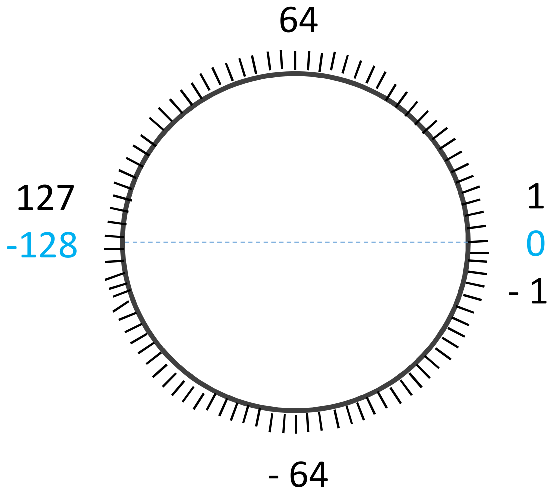
Recall: Overflow and Underflow e.g. `byte`

How is 127 represented as a `byte` ?

$(01111111)_2 = 127$

What happens if we add 1?

$(1000\ 0000)_2 = -128$



Overflow and Underflow e.g. `byte`

Recall that variables of type `byte` store values between -2^7 and $2^7 - 1$, that is, -128 and 127.

Overflow:

```
byte k = 127;
System.out.println(k+1);
System.out.println( (byte) (k+1));
```

Output:

128	(widening by promotion)
-128	(cast, narrowing)

Underflow:

```
byte j = -128;
System.out.println(j-1);
System.out.println((byte) (j-1));
```

Output:

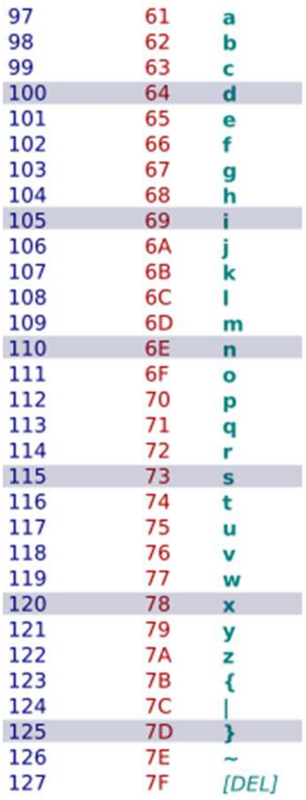
-129	**(widening by promotion)**
127	(cast, narrowing)

Examples of widening & narrowing `char`

```
char first = 'a';    // 97
char second = (char) (first + 1);
```

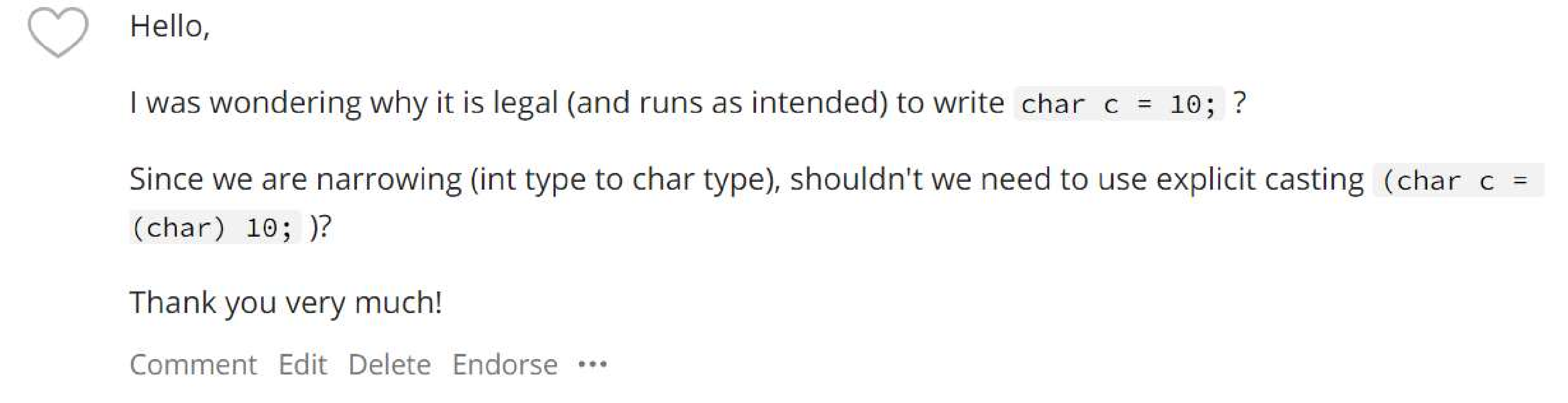
`first` is automatically converted into an `int` when performing `first + 1` , which evaluates to 98. (widening by promotion)

This `int` value is cast to `char` (narrowing), and `'b'` is stored in `second`.



97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	*[DEL]*

Posted late in course… followup



Hello,

I was wondering why it is legal (and runs as intended) to write `char c = 10;` ?

Since we are narrowing (int type to char type), shouldn't we need to use explicit casting `(char c = (char) 10;`)?

Thank you very much!

It seem Eclipse and IDEA don’t require an explicit down cast. Literals treated differently.

```
int i = 7;
char c1 = i; // compiler error
char c2 = 10; // no compiler error
```

ASIDE: Examples with `char` and `short`

```
char c = 'q';
short s = 2;
int   i = 3;
```

✓ allowed

`s =  i;` ✗ compile time error

`s = (short) i;` ✓

`s =  c;` ✗ compile time error

`s = (short) c;` ✓

`c =  s;` ✗ compile time error

`c =  (char)  s;` ✓

wider ↑

type	*number of bits*
double	64
float	32
long	64
int	32
char	16
short	16
byte	8

narrower ↓

Examples with `float` and `double`

`double y = 1/4;`		assigns value 0.0 to y
`double x = 1;`		legal, but considered bad style
`float y = 3.0;`	✗	compiler error
`float y = (float) 3.0;`	✓	narrowing
`float z =  3.0f;`	✓	

Coming up...

Lectures

Fri. Jan 14

Java Overview (JRE, JDK, ...)

Next week

arrays, strings, objects & classes

Homework (TODO)

- w3schools Tutorial (this week!)
- Install either Eclipse or IntelliJ. (this week!)
- Content -> tutorials.

  Tutorial (tomorrow)
- TA office hours